

# A Hardware Pipeline for Function Optimization using Genetic Algorithms

Malay K. Pakhira<sup>\*</sup>  
Kalyani Government Engineering College  
Kalyani - 741 235  
West Bengal, INDIA  
malay\_pakhira@yahoo.com

Rajat K. De<sup>†</sup>  
Indian Statistical Institute  
Kolkata - 700 108  
West Bengal, INDIA  
rajat@isical.ac.in

## ABSTRACT

Genetic Algorithms (GAs) are very commonly used as function optimizers, basically due to their search capability. A number of different serial and parallel versions of GA exist. In this paper, a pipelined version of the commonly used Genetic Algorithms and a corresponding hardware platform is described. The main idea of achieving pipelined execution of different operations of GA is to use a stochastic selection function which works with the fitness value of the candidate chromosome only. The modified algorithm is termed PLGA (Pipelined Genetic Algorithm). When executed in a CGA (Classical Genetic Algorithm) framework, the stochastic selection gives comparable performances with the roulette-wheel selection. In the pipelined hardware environment, PLGA will be much faster than the CGA. When executed on similar hardware platforms, PLGA may attain a maximum speedup of four over CGA. However, if CGA is executed in a uniprocessor system the speedup is much more. A comparison of PLGA against PGA (Parallel Genetic Algorithms) shows that PLGA may be even more effective than PGAs. A scheme for realizing the hardware pipeline is also presented. Since a general function evaluation unit is essential, a detailed description of one such unit is presented.

## Categories and Subject Descriptors

I.5 [Computing Methodologies]: Pattern Recognition;  
I.5.5 [Implementation]: Special Architecture—*Pipelining, performance measures*

---

<sup>\*</sup>M. K. Pakhira is associated to Department of Computer Science and Engineering at Kalyani Government Engineering College.

<sup>†</sup>R. K. De is associated to Machine Intelligence Unit at Indian Statistical Institute.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO '05, June 25–29, 2005, Washington, DC, USA  
Copyright 2005 ACM 1-59593-010-8/05/0006 ...\$5.00.

## General Terms

Algorithms, Design

## Keywords

Function optimization, Genetic algorithms, Hardware pipeline, Pipelined GA, Stochastic selection.

## 1. INTRODUCTION

Genetic algorithm (GA), developed by Holland in 1975 [8], is known to be an efficient search and optimization mechanism which incorporates the principles of evolution and natural selection. Many different forms of the basic GA exist [5]. Several attempts have been made to exploit the inherent parallelism of GA [7, 9, 6, 1, 11, 3]. Most of these methods maintain the basic serial nature of the operations. They simply divide the population into a number of sub-populations and execute genetic operations for each of the sub-populations separately. After parallel executions in all the processing units, at certain intervals, the newly developed information regarding the best solution strings of these units are exchanged through migration phase. This kind of parallel execution is supported by distributed processing environments. Recently, pipelined execution schemes for GAs are also being proposed [2, 10]. An attempt of designing a systolic array pipeline of the genetic operators may be found in [2].

In this paper, we describe the design of a pipelined genetic algorithm (PLGA) which uses a stochastic selection scheme that eliminates the dependency for a complete pool of candidate solutions required in conventional methods at the selection stage and hence allows us to develop a pipeline using the basic operations of GA. Simulation experiments with various functional optimization problems of varying complexity determines the applicability and effectiveness of this selection scheme in GAs. The performance of PLGA, in terms of rate of convergence and speedup, are found to be superior or comparable with respect to CGA and PGA. Since the full benefit of a pipelined GA can be realized only when it is executed on a complete hardware platform, we have presented a scheme for implementing the corresponding hardware pipeline. We have considered only binary representation of chromosomes in our design and for conducting simulation experiments. However, the hardware is highly general and can be used for optimization of any general function using PLGA.

## 2. BASICS OF GA

In genetic algorithms [8, 5], the parameters of an optimization problem corresponding to a possible solution are encoded to form a chromosome. A collection of such chromosomes is called a population or pool. The initial population is generated randomly or using some domain specific knowledge. Four different characteristic operations, viz., selection, crossover, mutation and evaluation are normally performed in GAs. The selection operator finds the fittest candidate solutions (chromosomes) from the pool of current generation, which are then represented in the next generation, based on their figure of merit. Classical GAs (CGA) generally use the roulette-wheel selection scheme. By crossover, features of two selected strings (mates) from the parent population are intermixed to generate newer chromosome patterns called children. Mutation is used for fine tuning the solution. Crossover and mutation are done with certain probabilities. Mutation can take place in several different ways. In the binary representation for chromosomes, it generally toggles one or more bit position(s) depending upon mutation probability. Using these three operators, we can get a new population from the old one. Crossover and mutation strategies used in this work are similar to those mentioned in [4]. By evaluation operation the objective function of the concerned optimization problem is evaluated. Since, in GAs, each of the chromosomes represents a probable solution, they all are evaluated during every generation to enumerate their fitnesses.

Parallel versions of GAs (PGA) are executed on a network of processors simultaneously. Difference among various parallel implementations are basically on the architecture of processor network, population distribution policy, rate of information exchange etc. We, however, have used a completely connected network in our experiments.

## 3. STOCHASTIC SELECTION AND PLGA

In this section, we shall describe a stochastic selection method which allows us to develop a pipeline of GA operators. This selection method is a modified version of the existing SA-selection [12]. The functional form of the selection method is described along with an example. Then we shall provide the corresponding pipeline design of GA.

In *stochastic* selection method, a chromosome  $\mathbf{x}$  is considered from a pool  $P(g)$  of the current generation  $g$ , and is selected based on Boltzmann probability distribution function.

Let,  $f_{max}$  be the fitness value of the currently available best string. If the next string is having fitness value  $f_{\mathbf{x}}$  such that  $f_{\mathbf{x}} > f_{max}$ , then it is selected. Otherwise, it is selected with Boltzmann probability

$$P = \exp[-(f_{max} - f_{\mathbf{x}})/T]$$

where,  
and

$$T = T_0(1 - \alpha)^k$$

$$k = 100 \times \frac{g}{G}$$

$G$  is the maximum value of  $g$ . Ranges of  $\alpha$  and  $T_0$  are  $[0, 1]$  and  $[5, 100]$  respectively. From the above expression, it is clear that the value of  $T$  will decrease exponentially or at logarithmic rate with increase in  $g$ , and hence the value of the probability  $P$ . This is significant in terms of convergence. As  $T \rightarrow 0$ , the final state is approached.

In conventional selection schemes, before starting the se-

**Table 1: Sample execution of stochastic selection**

Input chrom. number	Fitness of input chrom.	$f_{max}$	Chrom. number with maximum fitness	P	P1	Selected chrom. number
0	45.0	45.0	0	0.931	0.882	0
1	48.0	45.0	0	-	-	1
2	35.0	48.0	1	0.810	0.853	1
3	43.0	48.0	1	0.616	0.447	3
4	55.0	48.0	1	-	-	4
5	12.0	55.0	4	0.317	0.591	4

lection process, all the chromosomes in the earlier generations must be evaluated. But evaluation is the most time consuming process and is a bottleneck in attaining a pipeline of the operations of GAs. The new selection scheme eliminates this bottleneck. We can express the new selection operator as a function of the input chromosome ( $\mathbf{x}$ ). Let,  $\mathbf{x}_{max}$  be the chromosome corresponding to the currently available maximum fitness. Then the selection operator, expressed functionally, is

$$Sel(\mathbf{x}) = \begin{cases} \mathbf{x} & \text{if } (f_{\mathbf{x}} > f_{max}) \\ \mathbf{x} & \text{if } (f_{\mathbf{x}} \leq f_{max}) \wedge (P > P_1) \\ \mathbf{x}_{max} & \text{if } (f_{\mathbf{x}} \leq f_{max}) \wedge (P \leq P_1) \end{cases}$$

where,  $P_1 = random[0, 1)$ .

Let us consider an example, for describing the operation of the selection scheme, with a population of size 6. Let, in the  $(g - 1)$ th generation, the maximum fitness value is 45.0. This value is stored in the variable  $f_{max}$  and is used in generation  $g$  also. In any generation, the value of  $f_{max}$  is altered whenever a chromosome with a greater fitness is encountered (and selected). Note that, using elitist strategy, we are storing the best chromosome, in a generation, along with its fitness value in the very first location of the pool of chromosomes. Table 1 shows how chromosomes are selected for generation  $g$ .

A pair of selected chromosomes may be used for crossover, mutation, and evaluation and then put into the population pool for the next generation. When a chromosome is evaluated, it is put into population pool along with its fitness value for the next generation. Thus the processes corresponding to selection, crossover, mutation and evaluation in a particular generation can work simultaneously, in an overlapped fashion. Using the above mentioned selection method, the concept of pipelining has been incorporated within the genetic algorithm framework. It should be mentioned here that the SA-selection can as well be used for this purpose. The algorithmic difference of PLGA with respect to CGA is only due to the use of the stochastic selection scheme. However, since each chromosome can be selected (or rejected) as soon as it is evaluated, we can continue looping through the steps of four characteristic operations, whenever we have a pair of evaluated chromosomes in the population pool. The basic pipelined algorithm is shown in Algorithm 1. In this algorithm, the initial population is assumed to be pre-evaluated, before the process loops through the pipeline stages. However, other possibilities, like initializing the fitnesses randomly or rearranging the stages of the pipeline to perform evaluation before selection are possible.

**Algorithm 1 :** *Basic steps of Pipelined genetic algorithm.*

```

begin
  initialize population
  evaluate population for fitness
  while (NOT termination_condition) do
    begin
      select a pair of chromosomes
      cross the pair of chromosomes
      mutate the pair of chromosomes
      evaluate children for fitness
      put children into next generation pool
        with fitness values
    end
  end
end

```

Here, the processes corresponding to selection, crossover, mutation and evaluation in a particular generation can work simultaneously, in an overlapped fashion. It is also interesting to note that the generations are also overlapped. However, we maintained all the basic features of a generational GA during pipelined execution. This leads to reduction of appreciable amount of execution time as compared to conventional GA. The pseudo code showing the streamlined operations of selection, crossover, mutation and evaluation, within a generation, is given in Algorithm 2 below.

**Algorithm 2 :** *Outline of Pipelined genetic algorithm.*

```

begin
  g = 0;
  create pre-evaluated initial pool P(g) and
    initialize temperature T;
  repeat
    for i = 1 to Population_Size in steps of 2 do
      begin
        select a pair of chromosomes from pool P(g);
        cross the selected pair;
        mutate the crossed pair;
        evaluate the mutated pair and put in next
          generation pool P(g + 1);
      end
      g = g + 1;
      lower temperature T;
    until convergence;
  end
end

```

It is possible to implement the algorithm using *appropriate hardware*, where, selection, crossover, mutation and evaluation operations will be done in specially designed hardware circuits forming a pipeline among them. Multiplicity of processing units to be used at each of the stages may, in general, be different and depends on the complexity of the problem concerned.

## 4. DESIGN OF THE PIPELINE

In this section, we explore the structural parallelism of GA that are hidden in its strictly serial use of its different operations discussed in Section 3. For this purpose, we have

to streamline these operations so that their functioning becomes overlapped in nature. The motivations behind PLGA are twofold. First, there is an advantage in terms of higher speedup as a result of overlapped execution in a pipeline. The second, possibly more important one, is the chance of incorporating more population and thus increasing the diversity among them.

### 4.1 Pipeline Architecture

In order to solve functional optimization problems using GA, we need to maintain a population of probable solutions (chromosomes). The chromosomes are evaluated for the objective function. The fittest candidates are selected for the next generation which then undergo the crossover and mutation operations to generate offspring. The whole process is repeated for a number of generations. Thus, we can identify four major functions : (i) selection (S), (ii) crossover (C), (iii) mutation (M) and (iv) evaluation (E), and can construct a four stage pipeline as shown in Figure 1. Stage multiplicity requirements at mutation and evaluation stages are dependent on the complexity of the concerned problem and may be arranged by placing the corresponding units in parallel as shown in Figure 2. We need to maintain a buffer memory to reserve the chromosomes after evaluation. This buffer is organized in a FIFO manner.

The selection operation requires two parallel units so that it can provide two strings to the crossover unit in due time. Mutation and fitness evaluation should be done in multiple units that operate in parallel. The number of units for mutation is determined by the length of a chromosome. It is found that evaluation is the most time consuming process compared to the other operations, and the number of units for this stage is determined by the complexity of the function considered.

Let,  $S_t$ ,  $C_t$ ,  $M_t$  and  $E_t$  be the stage times for selection, crossover, mutation and evaluation operations respectively. Among them,  $C_t$  is normally found to be the minimum. We call this minimum time as one  $T$ -cycle. Let,

$$S_t = sC_t, M_t = mC_t \text{ and } E_t = eC_t.$$

Therefore, the ratio of  $S_t$ ,  $C_t$ ,  $M_t$  and  $E_t$  becomes  $s : 1 : m : e$ . That is,  $s$ ,  $m$  and  $e$  number of  $T$ -cycles are required for selection, mutation and evaluation operations respectively. Thus for one crossover unit we need, for efficient utilization of resources,  $[s]$ ,  $[m]$  and  $[e]$  pairs of units for selection, mutation and evaluation respectively. For sake of simplicity, let us consider, from now on,

$$s = [s], m = [m] \text{ and } e = [e].$$

Here the units are counted in pairs because one crossover needs two selected strings. From the above ratio, it is clear that, if the crossover unit takes 1 unit of time ( $T$ -cycle) to perform one crossover, the selection, mutation and evaluation units take  $s$ ,  $m$  and  $e$  units of time to perform one selection, mutation and evaluation operations respectively. Thus for proper and efficient utilization of the resources, we should use  $s$ ,  $m$  and  $e$  pairs of respective units for one crossover unit. Since, for complicated problems, the value of  $e$  may be very large compared to values of  $s$ ,  $m$  and  $e$ , it is necessary to design a simple but fast evaluation unit. In this paper, we have shown how such an evaluation unit can be achieved.

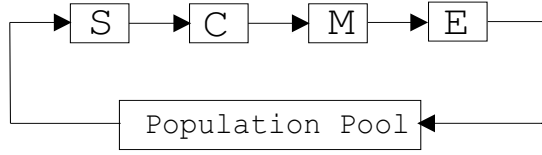


Figure 1: Pipeline stages for the GA. Here S, C, M and E stand for selection, crossover, mutation and evaluation respectively

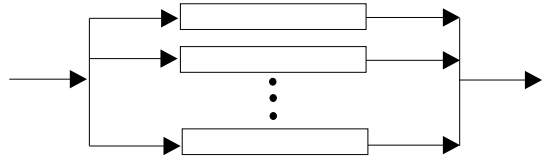


Figure 2: Multiplicity of any particular unit in the pipeline

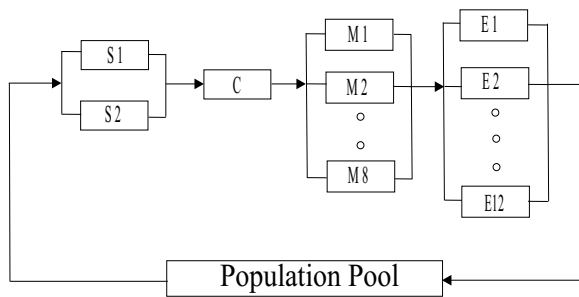


Figure 3: An example pipeline using different multiplicity at selection, crossover, mutation and evaluation stages

## 4.2 Speedup

Speedup of a general pipeline is defined as

$$S = \frac{T_{NP}}{T_P}$$

where  $T_{NP}$  ( $= nk$ ,  $n$  = number of executions and  $k$  = number of stages) and  $T_P$  ( $= n + k - 1$ ) are the computation times (in terms of number of  $T$ -cycles) required for non-pipelined and pipelined systems respectively. In the proposed clustering, if appropriate number of units are considered, the average time per chromosome in each stage becomes equal to one  $T$ -cycle. This is the ideal hardware configuration. However, we can use less number of units at the stages. Let, for any arbitrary configuration,  $m_n$  and  $e_n$  be the number of pairs of units used at mutation and evaluation stages corresponding to one crossover unit and one pair of selection units. In our case,

$n$  = population size  $\times$  number of generations.

Consider a pipeline where  $s = 1$ ,  $m = 4$  and  $e = 6$ .

Here we require  $s + 1 + m + e = 12$   $T$ -cycles to get the first pair of children chromosomes.

Since  $n$  = population size  $\times$  number of generations, we may assume, without loss of generality,  $n$  to be equal to the population size executed for one generation only. After obtaining the first pair of children, the remaining children will come out in pairs at each successive  $T$ -cycles. Therefore, the number of  $T$ -cycles required for the remaining pairs is  $(\frac{n}{2} - 1)$ . Thus, the total number of  $T$ -cycles required for the pipeline is

$$T_P = 11 + \frac{n}{2}.$$

For a non-pipelined system configured with the same multiplicity of stages (as that of the pipelined one), the number of  $T$ -cycles considering all the four stages sequentially is

$$T_{NP} = \frac{n}{2} + \frac{n}{2} + \frac{n}{2} + \frac{n}{2} = \frac{4n}{2} = 2n$$

So, the speedup attained is

$$S = \frac{T_{NP}}{T_P} = \frac{2n}{\frac{n}{2} + 11}$$

when  $n \gg 11$ ,  $S \approx 4$ . This is the ideal speedup.

We can use less number of units at mutation and evaluation stages. Let, for any arbitrary configuration,  $m'$  and  $e'$  be the number of pairs of units used at mutation and evaluation stages corresponding to one crossover unit and one pair of selection units. Here,  $m' < m$  and  $e' < e$ , i.e., the number of units at the mutation and evaluation stages are less than that needed for full multiplicity of these stages.

Let  $r_m = \lceil \frac{m'}{m} \rceil$  and  $r_e = \lceil \frac{e'}{e} \rceil$ ,

i.e.,  $r_m$  and  $r_e$  are the factors by which multiplicity is reduced at the corresponding stages.

We define the *reduction factor* for a pipeline as the maximum of  $r_m$  and  $r_e$ , i.e.,

$$\text{reduction factor, } r = \max(r_m, r_e).$$

When  $r = 1$ , the pipeline has full multiplicity and is referred to as a *full pipeline*. For  $r > 1$ , it is called a *reduced pipeline*.

Now, let us consider a reduced pipeline where  $r_m, r_e$  and  $r$  represent reduction factors for mutation, evaluation stages and that for the whole pipeline. By definition, at least one of  $r_m$  and  $r_e$  is equal to  $r$  and the other is less than or equal to  $r$ . For such a reduced system we get

$$T_P = 1 + 1 + m + e + \left(\frac{n}{2} - 1\right) \times r = 2 + m + e + \left(\frac{n}{2} - 1\right) \times r$$

and

$$T_{NP} = \frac{n}{2} + \frac{n}{2} + \frac{n}{2} \times \left\lceil \frac{m}{m'} \right\rceil + \frac{n}{2} \times \left\lceil \frac{e}{e'} \right\rceil.$$

If the pipeline is a uniformly reduced one with  $r_m = r_e = r$ , we have,

$$T_{NP} = \frac{n}{2} + \frac{n}{2} + \frac{n}{2} \times r + \frac{n}{2} \times r.$$

Now, in our example system, if the reduction factor be  $r = 2$ , then we get,

$$T_P = 12 + \left(\frac{n}{2} - 1\right) \times 2 = 10 + n$$

and

$$T_{NP} = \frac{n}{2} + \frac{n}{2} + \frac{n}{2} \times 2 + \frac{n}{2} \times 2 = 3n.$$

Therefore, speedup  $S \approx 3$  for  $n \gg 10$ .

From the above discussion it is clear that a fixed hardware setup is also possible. However, in the present work, we have not performed experiments with a reduced pipeline.

## 5. HARDWARE IMPLEMENTATION

A scheme for executing commonly used GAs in a hardware pipeline and the corresponding pipelined algorithm are presented in the earlier section.

The presented schematic architecture is easy to realize in a physical hardware. We need to implement each individual units using appropriate digital components and establish communication paths among them to complete the implementation. A possible configuration of the hardware pipeline is shown in Figure 3. It is easy to understand that implementation of crossover and mutation units are very simple if we consider only binary chromosomes. We are assuming that a selection unit can be developed in hardware. Then, we are left with the most important task of designing a hardware evaluation unit. This unit should be general enough, so that it can execute any general objective function. A possible design of such a general purpose evaluation hardware is presented in the following subsection.

### 5.1 A possible hardware evaluation unit

As mentioned earlier, out of the four pipeline stages, the evaluation stage is the most important and complicated part. Hence, we put importance to designing a general purpose evaluation unit. We use the concept of postfix function evaluation in a stack based hardware. We assume that any general algebraic function can be converted to its postfix form a priori.

A postfix expression can be evaluated in a simple stack based machine. Stack based machine is a very simple arithmetic execution unit where the expression to be evaluated is

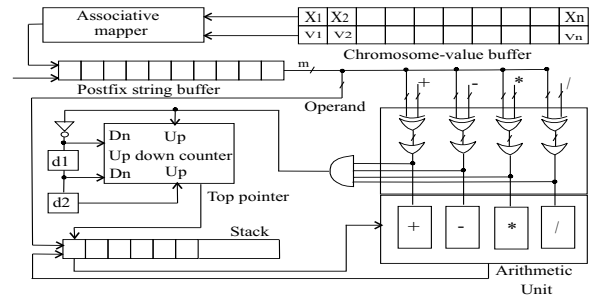


Figure 4: A sample hardware evaluation unit

stored in a shift register in postfix form. Then following the postfix evaluation algorithm we push operands, shifted out of the postfix register, onto a stack; and when an operator is encountered, we pop the two topmost operands from the stack and put them into the arithmetic unit and the result is pushed back onto the stack again. The design of a possible hardware evaluation unit is shown in Figure 4.

Here, the postfix expression is inputted in a right-shift register. Its contents are operands and operators only. A comparator circuit, consisting of multi-input XOR gates and OR gates, identifies the present input symbol from the postfix register. A stack is maintained as a combination of an LR-shift register and an up-down counter. The top of the stack is always at the leftmost position of the shift register.

Whenever an operand is detected by the comparator circuit, the up counter is incremented and the contents of the shift register is shifted one location toward right. The operand value is then pushed onto the stack (top).

When an operator is detected, the down counter is decremented and pop operation is performed by left shifting the shift register contents. The popped operands are passed to the arithmetic unit. After the operation is complete, the result from the arithmetic unit is again pushed onto the stack. It is to be remembered here that each time an operator is detected, two pop operations are needed consecutively which are needed for the operation. Thus the basic circuit presented here must be modified suitably. Also, it is assumed that all parts of the circuit operates with perfect synchronization. This may be achieved by use of delay elements at appropriate locations.

In our design, we have used two such delay elements  $d_1$  and  $d_2$ . The delay between the two pop operations is  $d_1$ . And after another delay of  $d_2$ , within which computation is done, the result is pushed back onto the stack by incrementing the up counter. Chromosomes are stored in a buffer called the chromosome-value buffer. Here both the symbolic chromosome and numeric values of its components are maintained. Let, the symbolic chromosome be denoted by  $X = x_1, x_2, \dots, x_n$  and the numeric chromosome be denoted by  $V = v_1, v_2, \dots, v_n$ . An associative mapping unit is used to substitute the numeric values for the chromosome symbols appearing in the postfix expression. After this substitution phase, the postfix expression contains numeric operands as well as operator symbols. To check whether an entry in the expression is an operand or an operator we use a simple logic circuit consisting of four multi-input XOR gates and four OR gates only. All entries are encoded in  $m$ -bits. When a match is found, one of the OR gate outputs will become zero, and so will be the external AND gate output. Then the

pop operation is started by decrementing the down counter. Otherwise a push operation would take place.

## 6. EXPERIMENTAL RESULTS

We have demonstrated effectiveness of PLGA on various benchmark functions. The benchmarks include both uni-modal and multi-modal functions. Experiments are performed to compare the performance of the pipelined algorithm (PLGA) with its serial (CGA) and parallel (PGA) counterparts. The following subsections describe the benchmark functions and results of comparison respectively.

### 6.1 Benchmark Functions

The selected benchmark functions, which are unimodal or multi-modal in nature, are available in literature. These functions are mentioned below.

1. *Sphere Model* function:

$$f_1(x) = \sum_{i=1}^l x_i^2.$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 0, \forall i$ .

2. *Step* function:

$$f_2(x) = \sum_{i=1}^l \text{integer}(x_i).$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 0, \forall i$ .

3. *Rastrigin's* function:

$$f_3(x) = \sum_{i=1}^l [x_i^2 - 10 \cos(2\pi x_i) + 10]$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This multi-modal function has its minimum value of 0 at  $x_i = 0, \forall i$ .

4. *Rosenbrock's* function:

$$f_4(x) = \sum_{i=1}^l \left[ 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 1, \forall i$ .

5. *Ackley's* function:

$$f_5(x) = -20 \exp \left( -0.2 \sqrt{\frac{1}{l} \sum_{i=1}^l x_i^2} \right) - \exp \left( \frac{1}{l} \sum_{i=1}^l \cos 2\pi x_i \right) + 20 + e$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 0, \forall i$ .

6. *Schwefel's* function 1:

$$f_6(x) = \sum_{i=1}^l |x_i| + \prod_{i=1}^l |x_i|$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 0, \forall i$ .

7. *Schwefel's* function 2:

$$f_7(x) = \sum_{i=1}^l \left( \sum_{j=1}^i x_j \right)^2$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 0, \forall i$ .

8. *Schwefel's* function 3:

$$f_8(x) = \max_i \{ |x_i|, 1 \leq i \leq l \}$$

The range of  $x_i$  is  $-5.12 \leq x_i \leq 5.12$ . This function has its minimum value of 0 at  $x_i = 0, \forall i$ .

For all these functions we have coded the variables using 25 bit binary code. However, larger binary codes may be used to increase the accuracy of the numbers represented.

## 6.2 Results

Results of experiments are described in two phases. In the first phase, we compared stochastic and roulette-wheel selection schemes in CGA framework, in terms of quality of solutions and rate of convergence and speedup. The speedup obtained in the pipelined system over that obtained with CGA executed on the same hardware platform and also for uniprocessor based CGA are presented. In the second phase, we demonstrate comparative performances of PLGA and a particular version of PGA in terms of rate of convergence.

### 6.2.1 PLGA vs. CGA

For this part of investigations, with the stochastic and roulette-wheel selection schemes in the CGA framework, we have used the same values for the control parameters, viz., population size = 50,  $P_c = 0.6$  and  $P_m = 0.05$ . The value of  $\alpha$  is taken to be 0.05. The initial temperature,  $T_0$ , is set to 50.

At first, we have compared the stochastic selection scheme with the roulette wheel one when both of them are used in a CGA framework. Then, we have executed the PLGA and CGA using the stochastic selection function in both the cases for computing speedup. Simulation experiments are performed on the selected problems mentioned in Section 6.1. We have used, here, a dimension of 10 for all the problems. Table 2 provides the optimal values for the objective functions which are reached by the CGA, using two selection schemes, when executed for 2000 generations.

Speedup of PLGA compared to CGA is measured in two different situations, viz., when a similar hardware platform (like the pipeline) is used for CGA, and when a single uniprocessor is used only. For the first situation, results are provided in Table 3, and for the second, it is provided in Table 4. The stage times given in Table 3 are proportional to the actual times. This is because, we have executed PLGA for a number of generations, and measured sum-total of the corresponding stage times. The speedup is found to be less than that obtained in Section 4.2. This is due to the fact that in computation of  $T_P$  in Section 4.2, we have assume extra time to all the units in order to achieve perfectly synchronous operation of all the pipeline stages. However, selection and crossover could be done in less amount of time than that allocated (50 units). If we allocate 25 units of times, for example, as the pipeline stage time, then speedup would also be improved.

**Table 2: Comparison of stochastic and roulette wheel selections. Results are averaged over 50 runs. “Mean Best” and “Std Dev” indicate mean best function values and standard deviations**

Function	Stochastic		Roulette wheel	
	Mean Best	Std Dev	Mean Best	Std Dev
$f_1$	$1.50 \times E - 6$	$1.28 \times E - 6$	$3.06 \times E - 6$	$1.37 \times E - 6$
$f_2$	0.00	0.00	0.00	0.00
$f_3$	2.5626	1.3813	4.0030	2.0641
$f_4$	8.6902	0.8009	34.7507	39.2635
$f_5$	$1.32 \times E - 3$	$4.30 \times E - 4$	$1.50 \times E - 3$	$6.08 \times E - 4$
$f_6$	$2.90 \times E - 3$	$6.50 \times E - 4$	$3.15 \times E - 3$	$9.42 \times E - 4$
$f_7$	$2.60 \times E - 3$	$4.09 \times E - 4$	$5.01 \times E - 2$	$7.85 \times E - 2$
$f_8$	$5.55 \times E - 3$	$1.68 \times E - 3$	$5.67 \times E - 3$	$1.27 \times E - 3$

**Table 3: Stage times of PLGA and speedups obtained for  $f_1 - f_8$ . Times (in units of  $10^4$  clock ticks) shown are proportional to actual stage times. One  $T$ -cycle = 50.**

Func	Dim	Stage Times				No. of $T$ -Cycles				Speedup
		$S_t$	$C_t$	$M_t$	$E_t$	$s$	$c$	$m$	$e$	
$f_1$	3	17	13	150	599	1	1	3	12	2.60
$f_2$	5	18	14	247	998	1	1	5	20	2.62
$f_3$	10	15	46	472	2129	1	1	10	43	2.90
$f_4$	3	14	19	134	539	1	1	3	11	2.53
$f_5$	3	16	18	134	527	1	1	3	11	2.53
$f_6$	3	16	10	137	481	1	1	3	10	2.40
$f_7$	5	16	23	216	842	1	1	5	16	2.70
$f_8$	3	14	15	138	482	1	1	3	10	2.46

**Table 4: Speedup of PLGA over CGA executed in a serial uniprocessor system with no special hardware processing elements. The times (in units of  $10^4$  clock ticks) shown are proportional to actual stage times. One  $T$ -cycle = 50.  $k = n/2$**

Function	Dimension	Total Execution Time		Speedup
		Serial Uniprocessor	Pipelined System	
$f_1$	3	779k	$(16+k)50$	15.58
$f_2$	5	1277k	$(26+k)50$	25.54
$f_3$	10	2662k	$(54+k)50$	53.24
$f_4$	3	706k	$(15+k)50$	14.12
$f_5$	3	695k	$(15+k)50$	13.90
$f_6$	3	644k	$(14+k)50$	12.88
$f_7$	5	1097k	$(22+k)50$	21.94
$f_8$	3	649k	$(14+k)50$	12.98

### 6.2.2 PLGA vs. PGA

We have executed both PLGA and PGA for a number of selected benchmark problems. In this case, again we have set the dimension of benchmark functions to 10. As a measure of comparison, we have selected the number of generations needed to converge to a near optimal solution. For all the benchmark functions, a particular limiting value is selected as the stopping criteria.

Here, the population size considered is 40. For the purpose of executing PGA a four processors network is considered. The population of size 40 is distributed among the four processors, each getting a subpopulation of size 10. The processors are completely connected and they can communicate strings (chromosomes) after every 5 generations. During communication, each processor selects four chromosomes, including the current best, from self, and two from each of the other processors. The results of comparison are shown in Table 5.

## 7. CONCLUSION

A function optimizer, using a pipelined version of the conventional genetic algorithm, called PLGA, and a corresponding hardware pipeline have been described in this paper. A stochastic selection scheme is used for this purpose. The

**Table 5: Comparison of PLGA and PGA in terms of number of generations. “Mean” and “Stddev” indicate the average number of generations and the standard deviations respectively.**

Function	Dimension	PLGA		PGA		Stopping Value
		Mean	Stddev	Mean	Stddev	
$f_1$	10	180.52	42.92	406.50	61.15	0.005
$f_2$	10	33.72	13.71	54.52	15.49	0.005
$f_3$	10	8.06	3.08	17.42	6.58	50.0
$f_4$	10	11.18	2.94	20.76	5.93	500.0
$f_5$	10	65.38	23.86	128.98	33.54	0.005
$f_6$	10	134.18	32.72	284.26	36.71	0.5
$f_7$	10	132.02	138.19	202.40	109.82	0.5
$f_8$	10	236.00	58.62	383.56	64.01	0.5

unique feature of the stochastic selection scheme is that, it does not depend on a complete pool of pre-evaluated chromosomes. Hence, as soon as a chromosome is evaluated, it can be passed to the selection unit for possible selection.

By use of proper hardware, one can develop an extremely fast version of the GA based function optimizer. One scheme of realizing the hardware evaluation unit for function optimization is presented along with a block diagram of the complete pipeline. However, we have executed the PLGA algorithm on a serial uniprocessor system, and have shown that, in a hardware implementation, if proper multiplicity of different stage units are used, a maximum speedup of 4 is attainable compared to conventional GAs executed serially using similar multiplicity of stage units. However, when compared to CGA executed on a uniprocessor, speedup is found to be much more. We have also compared PLGA and PGA with a certain processor architecture. It is seen that speedup obtained in PLGA is better than PGA.

Although PLGA is presented here only as a functional optimizer, one may use it for any combinatorial optimization problem also. The authors are currently working in that direction.

## 8. ACKNOWLEDGMENTS

This research is partly supported by a sponsored project titled *Pipelined Genetic Algorithm and its Applications in Satellite and Medical Image Segmentation* : Number 8022/RID/ NPROJ/ RPS-97/ 2003-04 funded by *All India Council for Technical Education (AICTE)*, Government of India.

## 9. REFERENCES

- [1] S. Baluja. Structure and performance of fine-grain parallelism in genetic search. In *Proc. of the Fifth International Conference on Genetic Algorithms*, pages 155–162, Morgan Kaufmann, San Mateo, CA, 1993.
- [2] I. M. Bland and G. M. Megson. Efficient operator pipelining in a bit serial genetic algorithm engine. *Electronic Letters*, 33:1026–1028, 1997.
- [3] E. Cantú-Paz. A survey of parallel genetic algorithms. Technical report, University of Illinois, Illinois GA Laboratory, Urbana Champaign, Urbana, IL, 1997.
- [4] J. L. R. Filho, P. C. Treleven, and C. Alippi. Genetic algorithm programming environments. *IEEE Computer*, pages 28–43, June, 1994.
- [5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, New York, 1989.

- [6] V. S. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proc. of the Fifth International Conference on Genetic Algorithms*, pages 177–183, Morgan Kaufmann, San Mateo, CA, 1993.
- [7] J. J. Grefenstette. Parallel adaptive algorithms for function optimization. Technical report, Vanderbilt University, Computer Science Department, Nashville, TN, 1981.
- [8] J. Holland. *Adaptation in Neural and Artificial Systems*. University of Michigan, Ann. Arbor, MI, 1975.
- [9] H. Mühlenbein, M. Scomisich, and J. Born. The parallel genetic algorithm as function optimizer. In *Proc. of Fourth Intl. Conf. on Genetic Algorithms*, pages 271–278. Morgan Kaufmann, San Mateo, Calif, 1991.
- [10] M. K. Pakhira. A hybrid genetic algorithm using probabilistic selection. *Journal of the Institution of Engineers (India)*, 84:23–30, 2003.
- [11] R. Shonkwiler. Parallel genetic algorithms. In *Proc. of 5th Intl. Conf. on Genetic Algorithms*, pages 199–205. Morgan Kaufmann, San Mateo, CA, 1993.
- [12] B. T. Zhang and J. J. Kim. Comparison of selection methods for evolutionary optimization. *Evolutionary Optimization*, 2(1):55–70, 2000.